Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Vid Juvan

Generičen gonilnik za serijsko komunikacijo

diplomsko delo

univerzitetni interdisciplinarni študijski program prve stopnje računalništvo in matematika

Mentor: izr. prof. dr. Branko Šter

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Raziščite možnost implementacije generičnega gonilnika, ki bi ga s čim manj potrebnega dela lahko pretvorili v gonilnik za poljubno specifično napravo. Kot primer vzemite gonilnike za različne vrste serijske komunikacije; napišite torej generičen gonilnik za serijsko komunikacijo. Opišite smisel, razvoj, delovanje in uporabo takega gonilnika.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Vid Juvan, z vpisno številko **63100171**, sem avtor diplomskega dela z naslovom:

Generičen gonilnik za serijsko komunikacijo

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Branka Štera,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 20. avgusta 2014 Podpis avtorja:

Kazalo

Povzetek

Namen diplomske naloge je najprej nazorno prikazati in razložiti delovanje serijske komunikacije ter pregledati trenutno stanje na tem področju. Drugi cilj diplomske naloge pa je ustvariti generičen gonilnik za serijsko komunikacijo.

Ker je potrebno za vsak tip naprave implementirati gonilnik, je smiselno poenostaviti implementacijo in skrajšati čas dela. Za te namene bi radi ustvarili generičen gonilnik za serijsko komunikacijo, ki bi ga s čim manj potrebnega dela lahko pretvorili v gonilnik za poljubno specifično napravo. Te cilje zajema dotična diplomska naloga.

Za pravilno implementacijo je potrebno jasno poznavanje same serijske komunikacije in vseh njenih aspektov. Diplomska naloga se ukvarja prav s temi vprašanji in predlaga rešitve ter optimizacije. Del naloge je tako namenjen tudi študiji serijske komunikacije, na podlagi katere temeljijo kasnejše odločitve in argumenti.

Diplomska naloga končno poda tudi podlago za nadaljnji razvoj.

Ključne besede: serijska komunikacija, gonilniki, generičen gonilnik.

Abstract

The goal of the thesis is first to provide a clear explanation of how serial communication works and to review the latest work and progress in the field of serial communication. The second goal of this thesis is to create a generic driver for the purposes of communicating with a device using a serial communication.

Since it is required for every type of a device that communicates using a serial port, to implement a dedicated driver, it is logical to simplify this implementation and shorten the working time as much as possible. To achieve this, we would like to develop a generic driver for serial communication, that could be converted to a dedicated driver for any specific device with the least possible effort. These are the ambitions of this particular thesis.

For a proper implementation, a clear and wide knowledge of the serial communication functionality and its aspects is required. The thesis oversees these problems and suggests solutions and optimizations. A part of the thesis is hence also intended for the study of serial communication, upon which are based subsequent decisions in the implementation of the generic driver.

The thesis finally provides a basis for future research.

Keywords: serial communication, drivers, generic driver.

Poglavje 1

Uvod

Namen dela je povzeti znanje na področju serijske komunikacije in razvijati programsko opremo na podlagi le tega. Ta zajema osnovne lastnosti, primerjavo z ostalimi vrstami računalniške komunikacije, primere uporabe, pristope programiranja, najnovejša dela in raziskave področja, študij in ovrednotenje obstoječe programske opreme. Nadalje to znanje vključuje tudi poznavanje serijske komunikacije na fizičnem nivoju, poznavanje standardov priključkov in povezovalnega medija, in varnost pri sami komunikaciji.

Na tej osnovi bomo v drugi fazi diplomske naloge oblikovali in ustvarili generičen serijski gonilnik. S pristopom generične implementacije želimo spodbuditi razvoj gonilnikov v tej smeri in tako optimizirati potek in čas implementacije gonilnikov.

Generičen gonilnik definirajmo kot gonilnik, ki pri pravilni konfiguraciji deluje kot specifičen gonilnik za katerokoli napravo. Strukturo takšnega generičnega gonilnika v grobem delimo na vsaj dva dela. Prvi del skrbi za dejansko komunikacijo, vse varnostne nastavitve in kakovost prenosa. Drugi del gonilnika zajema dejansko strukturo in format klicev in odgovorov. Ideja je, da bi za neko določeno napravo, s katero želimo komunicirati preko serijskega vhoda, generičnemu gonilniku podali samo konfiguracijo. Ta bi vključevala spisek vseh možnih ukazov, ki jih lahko pošlje, njihov format in format odgo-

vora, ki ga lahko prejme. Generičen gonilnik bi iz teh informacij skonstruiral vse potrebne funkcije, kjer bi vsaka funkcija pripadala enemu ukazu, in jih implementiral po navodilih formatov. S tako strukturo gonilnika bi olajšali pisanje specifičnih gonilnikov v prihodnje. Za katerokoli napravo bi morali napisati zgolj konfiguracijsko knjižnico, ki bi sovpadala z navodili serijske komunikacije te naprave v njenem priročniku. Tak generičen gonilnik služi kot orodje oziroma okvir za hitro implementacijo specifičnih gonilnikov, ki komunicirajo preko serijskega vhoda.

Razvoj na področju serijske komunikacije je pomemben zaradi njene uveljavljenosti. Ta se pogosto uporablja, kljub temu da jo pri delu z manjšimi vhodno-izhodnimi napravami dandanes večinoma nadomešča USB standard.

Prav tako je dobrodošla optimizacija implementacije programske opreme serijske komunikacije. Eden teh je implementacija gonilnikov, ki jo ima ta diplomska naloga namen pohitriti.

Poglavje 2

Pregled področja

Serijska komunikacija se dobro uveljavlja in pogosto uporablja kot sredstvo komunikacije. Na nekaterih področjih jo izrivajo drugi načini komunikacije, v drugih primerih pa se je serijska komunikacija izkazala kot najboljša rešitev. Različne študije se ukvarjajo s serijsko programsko tehnologijo v različnih sistemih, kjer ravno komunikacija predstavlja največji problem [11] [6]. Izsledki drugih raziskav temeljijo na znanju in uporabi tehnologij dosedanjega razvoja za oblikovanje prihodnjih fizičnih produktov [1]. Študije se ukvarjajo tudi s problematiko same implementacije gonilnikov naprav [10]. V razvoju so nove tehnologije in izboljšave [5]. Te izboljšujejo kakovost in hitrost komunikacije. Druge raziskave se ukvarjajo s podporo za bolj specifične razširitve funkcionalnosti [2]. Nadalje študij zavzema tudi dodatno specifično implementacijo funkcionalnosti nad že ustaljeno delovanje komunikacije za dosego specifično določenih ciljev [4]. Nekatere raziskave se neposredno ukvarjajo z razvojem novih načinov, zasnov in implementacij serijske komunikacije, ki rešujejo probleme upravljanja kompleksnih komunikacijskih protokolov in hkrati zagotavljajo učinkovito, stabilno in zanesljivo komunikacijo [8]. Pojavljajo se različni načini implementacije serijske komunikacije [9], ki pa v večji meri sledijo uveljavljenim pristopom programiranja [7].

Vsekakor je jasno razvidno, da se bo razvoj na tem področju še širil in razvijal tudi v bodoče, potrebe po novih rešitvah in želje po dodatnih

aplikacijah pa bodo vedno prisotne.

Generičen gonilnik se pojavlja v nekaterih raziskavah računalniških komunikacij [3], vendar se te neposredno ne nanašajo na samo implementacijo gonilnika ali njegovo uporabo oziroma razširitev. To področje je slabše zastopano kot sama serijska komunikacija.

Poglavje 3

Serijska komunikacija

Serijska komunikacija je proces pošiljanja podatkov preko električne linije. Ponavadi poteka med računalnikom in periferno napravo. V kontrastu s paralelnim komuniciranjem, se podatki pri serijskem prenašajo en bit naenkrat. Komunikacijski medij lahko zato zajema manj tokov. Ker se prenaša naenkrat samo en bit, je hitrost prenosa na en urin takt manjša, vendar enostavnejša struktura omogoča hitrejši urin takt. Ker tako razlike v hitrosti med paralelnim in serijskim komuniciranjem niso prevelike in ker je implementacija serijske komunikacije cenejša, je ta zato tudi bolj pogosto uporabljena.

Za komunikacijo med dvema napravama je v osnovi potreben medij, po katerem potujejo signali, oddajnik in prejemnik signalov na obeh napravah ter pravilo oziroma protokol komunikacije. Vsi deli komunikacije morajo sovpadati. Za sporazum naprav pri komunikaciji obstajajo standardi, ki določajo tip priključka, vrsto medija, električne karakteristike, časovno odvisnost signalov, pomen vsakega od signalov in s tem način sporazumevanja naprav.

3.1 Standardi serijske komunikacije

Elektronsko komunikacijo podatkov na splošno delimo v dve skupini: enojno(angl. single-ended) in diferencialno (angl. differential).

Enojna komunikacija je preprosta in najpogosteje uporabljena metoda pošiljanja električnih signalov preko komunikacijskih kanalov, v večini primerov žic. Za enojno povezavo zadostujeta že dva povezovalna kanala. Prvi je napetostni oziroma signalni kanal, ki povezuje vir signala in mesto podatkovnega prevzema. Variacije v napetosti tega kanala predstavljajo podatkovno sporočilo. Drugi povezovalni kanal je priključen na referenčno napetost, najpogosteje zemljo, in se uporablja kot osnova za primerjavo napetosti. Rezultat predstavlja razlika v meritvi napetosti signalnega in referenčnega kanala. Za širitev širine komunikacije zadostuje dodaten signalni kanal. Za n signalov potrebujemo $n + 1$ kanalov, torej za vsak signal svoj kanal in en dodaten kanal za referenčno napetost. Prednost enojne metode serijske komunikacije je poceni realizacija in malo število povezovalnih kanalov na signal. Slabosti te metode pa predstavlja slaba odpornost na šum, ki ga povzročijo spremembe napetosti zemlje med oddajnim in sprejemnim vezjem ter indukcija, ki vpliva na žico signalnega kanala. Te slabosti odpravlja diferencialna metoda komunikacije.

Diferencialni način signalizacije je metoda pošiljanja signala preko dveh komplementarnih signalnih kanalov oziroma žic, ki jima pravimo diferencialni par. Signal v normalni obliki potuje po enem izmed kanalov, pa drugem pa potuje komplementarna verzija istega signala. Rezultat signala na prejemnikovi strani predstavlja razlika v napetosti na signalnih kanalih. Napetost zemlje je tukaj irelevantna in tako ne povzroča motenj. Glavna prednost diferencialnega načina prenosa signala je odpornost na elektromagnetne motnje, imenujemo jih presluh (angl. crosstalk), ki inducirajo šum na signalnih kanalih. Zunanji vplivi delujejo na obe žici diferencialnega para enako, s čimer se ohranja razlika v napetosti med njima. Ker je informacija na kanalih predstavljena samo z razliko v napetosti, je komunikacija pri diferencialnem načinu signalizacije načeloma odporna na zunanje elektromagnetne motnje. Dife-

rencialni par žic je dodatno še prepleten v parico. To zmanjšuje inducirano napetost. Poleg tega je posledično kakršenkoli zunanji vpliv enakomerno porazdeljen med diferencialni par. Slabost diferencialnega načina signaliziranja je dražja realizacija in večje število povezovalnih kanalov na signal. Za n signalov namreč potrebujemo vsaj $2n$ kanalov.

Najpogostejši standardi:

- RS-232
- RS-422
- RS-423
- RS-485

V tabeli 3.1 je podana specifikacija pogostejših standardov serijske komunikacije.

Tabela 3.1: specifikacija standardov serijske komunikacije

SPECIFIKACIJE	RS232	RS422	RS423	RS485
Način delovanja	ENOJNI	DIFER.	ENOJNI	DIFER.
Število oddajnikov na liniji	1	1	1	32
Število prejemnikov na liniji	1	10	10	32
Dolžina kabla	15m	1500m	1500m	1500m
Hitrost prenosa podatkov	20kb/s	10Mbit/s	10Mb/s	10Mb/s
Izhodna napetost oddajnikov	+/-25V	+/-6V	-0.25V, 6V	-7V, +12V

3.1.1 Standard RS-232

Standard RS-232C (večinoma imenovan RS-232) je pogosto uporabljen standard serijske komunikacije za prenos podatkov na kratke razdalje. V preteklosti je bil skoraj vsak računalnik opremljen z enimi ali več serijskimi vrati

(port) in gonilniki za komunikacijo preko RS-232 vrat. Uporaba tega standarda močno upada. Pri povezavi s perifernimi napravami ga nadomešča komunikacijski standard USB. V večini primerov se standard RS-232 uporablja pri povezavi DTE elementa (Data Terminal Equipment), kot je računalnik, z DCE (Data Circuit-terminating Equipment) elementom, kot je modem. RS-232 uporablja enojni način komunikacije. Je zelo občutljiv na interferenco zunanjih signalov, zato je dolžina komunikacijskega kabla omejena na 15m. RS-232 se uporablja samo za dvotočkovno povezavo (en pošiljatelj, en prejemnik). Simultano pošiljanje večje električne moči preko tega standarda načeloma ni mogoče. Uporablja se samo za napajanje naprav, ki imajo zelo nizko porabo električne energije.

RS-232 priključek

Za standard RS-232 se večinoma uporabljata priključka z 9 ali 25 pini "DE-9"in "DB-25". Moški priključek se uporablja na strani DTE, ženski priključek pa na strani DCE.

Na sliki 3.1 in v tabeli 4.1 je podana specifikacija in razporeditev pinov DE-9 priključka za standard RS-232.

Slika 3.1: Priključek DE-9

3.1.2 Standard RS-422

Standard RS-422, kvalificiran pod imenom TIA/EIA-422 velja kot standard visoke hitrosti. Uporablja diferencialen način prenosa podatkov. Standard omogoča način delovanja z do deset prejemniki signala (angl. multi-drop).

Tabela 3.2: specifikacija pinov RS-232 DE-9

1	DCD	Data Carrier Detect	6	DSR	Data Set Ready
2	RD	Received Data	7	RTS	Request To Send
3	TD	Transmitted Data	8	CTS	Clear To Send
4	DTR	Data Terminal Ready	9	RI	Ring Indicator
5	GND	Signal Ground			

Standard je primeren za prenos podatkov tudi na daljše razdalje. Omogoča hitrost do 10Mbit/s in dolžino povezovalnega medija do 1500m.

RS-422 priključek

Za standard RS-422 se večinoma uporabljata priključka z 9 ali 37 pini "DE-9"in "DB-37". Moški priključek se uporablja na strani DTE, ženski priključek pa na strani DCE.

Na sliki 3.2 in v tabeli 3.3 je podana specifikacija in razporeditev pinov DE-9 priključka za standard RS-422.

Slika 3.2: Priključek DE-9

Tabela 3.3: specifikacija pinov RS-422 DE-9

1	TXD-	Transmitted Data -	6	RXD-	Received Data -
2	TXD+	Transmitted Data +	7	RXD+	Received Data +
3	RTS-	Request To Send -	8	CTS-	Clear To Send -
4	RTS+	Request To Send +	9	CTS+	Clear To Send +
5	GND	Signal Ground			

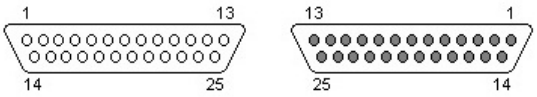
3.1.3 Standard RS-423

Standard RS-423 je nepogosto uporabljen standard. Gre za nadgrajeno različico standarda RS-232, ki se lahko uporablja tudi za prenos na dajše razdalje. Vmesnik simulira uporabo diferencialnega način pošiljanja. Vsi pini, ki naj bi služili drugemu signalu, pa so v bistvu priklopljeni na zemljo. RS-422 se lahko za razliko od RS-232 uporablja tudi za večtočkovno komuniciranje (en pošiljatelj, do deset prejemnikov).

RS-423 priključek

Za standard RS-423 se večinoma uporabljata priključka s 25 ali 37 pini "DB-25"in "DB-37". Moški priključek se uporablja na strani DTE, ženski priključek pa na strani DCE.

Na sliki 3.3 in v tabeli 3.4 je podana specifikacija in razporeditev pinov DB-25 priključka za standard RS-423.



Slika 3.3: Priključek DB-25

Tabela 3.4: specifikacija pinov RS-423 DB-25

1	GND	8	RLSDA	14	TDB	20	DTRA
2	TDA	9	RTA	15	TTA	21	RL
3	RDA	10	RLSDB	16	RDB	22	DSRB
4	RTSA	11	XTB	17	RTA	23	DTRB
5	CTSA	12	TTB	18	LL	24	XTA
6	DSRA	13	CTSB	19	RTSB	25	TM
7	GND						

3.1.4 Standard RS-485

Standard RS-485 je eden najbolj izpopolnjenih standardov. Uporablja diferencialen način. Standard se lahko uporablja za prenos informacij tudi na daljše razdalje. Primeren je za dvotočkovno in večtočkovno(do 32 pošiljateljev in 32 prejemnikov) komunikacijo. Hitrost prenosa dosega 35 Mbs.

RS-485 priključek

Za standard RS-485 se večinoma uporablja priključek z 9 pini "DE-9". Moški priključek se uporablja na strani DTE, ženski priključek pa na strani DCE.

Na sliki 3.4 in v tabeli 3.5 je podana specifikacija in razporeditev pinov DE-9 priključka za standard RS-485.

Slika 3.4: Priključek DE-9

Tabela 3.5: specifikacija pinov RS-485 DE-9

1	GND	Common Ground	6	CTS-	Clear To Send -
2	CTS+	Clear To Send +	7	RTS-	Request To Send -
3	RTS+	Ready To Send +	8	TXD+	Transmitted Data +
4	RXD+	Received Data +	9	TXD-	Transmitted Data -
5	RXD-	Received Data -			

3.2 Konfiguracija serijske komunikacije

Vsak pogovor serijske komunikacije mora biti pravilno oziroma enako konfiguriran na obeh straneh. V nadaljevanju sledi opis osnovnih in najpomemb-

nejših komponent konfiguracije serijske komunikacije.

Serijska komunikacija je realizirana s pošiljanjem in prejemanjem nizov informacij, ki mu pravimo okvir (angl. character frame). Okvir je sestavljen iz začetnega bita, podatkovnih bitov, parnega bita (ali brez) in končnih bitov. Uporablja se notacija D/P/S, ki označuje Podatek(Data) / Parnost(Parity) / Konec(Stop). Bolj pogosta uporaba je 8/N/1, ki določa uporabo 8 podatkovnih bitov, onemogočen parni bit in 1 končni bit. Notacija 7/E/1 določa 7 podatkovnih bitov, uporabo sodega (even) parnega bita in 1 končni bit. Za določitev lihega (odd) parnega bita se uporablja notacija O.

3.2.1 Baudova hitrost (Baud rate)

Baudova hitrost (Baud rate) označuje maksimalno hitrost prenosa podatkov v simbolih na sekundo. V večini primerov baud rate sovpada z bit rate oznako, vendar pojma nista identična. Baud rate označuje hitrost spremembe iz ene faze v drugo. Vsaka faza je lahko določena z večimi biti. V primeru, ko ločimo 4 faze napetosti, je vsaka faza predstavljena z 2 bitoma. V takem primeru baud rate 4800 pomeni hitrost 9600 bit/s. Baud rate poleg podatkovnih bitov zajema tudi vse ostale bite pri prenosu, to so začetni biti, parni bit in zaključni biti. V primeru kvalificiranja v dve fazi napetosti, če uporabljamo en začetni bit, parni bit in dva zaključna bita, se pri hitrosti prenosa 19200 baud prenese $\frac{19200}{1+8+1+2} = 1600$ bajtov podatkov na sekundo.

3.2.2 Parni bit (Parity bit)

Uporaba parnega bita je najosnovnejša metoda zaznavanja napak pri prenosu. Parni bit je dodaten bit, ki je lahko prisoten na koncu podatkovnega dela vsakega prenesenega seta, pred končnimi biti. Označuje, ali je število enic v binarnega zapisu prenesenega seta sodo, ali liho. Poznamo dve variaciji: sodi parni bit in lihi parni bit. Pri uporabi sodega parnega bita se prešteje vse enice v setu. Če je to število liho, se parni bit postavi na 1, kar pomeni da je število enic v setu, ki vključuje obravnavani prani bit, sedaj

soda. Obratno velja pri uporabi lihega parnega bita. Pri sprejemu sporočila se število enic v vsakem setu ponovno prešteje in preveri sodost. Pri ugotovitvi, da parnost ne drži, je prišlo do napake pri prenosu. V takem primeru prejemnik najpogosteje obvesti odajnika signala o nastali napaki, nakar sledi ponovno pošiljanje sporočila. Parni bit je lahko prisoten, vendar ne nujno uporabljen. Če je vedno postavljen na vrednost 1, temu pravimo Mark parni bit. V primeru, ko je vedno enak 0, mu pravimo Space parni bit.

3.2.3 Podatkovni biti (Data bits)

Število podatkovnih bitov označuje dolžino bitnega zapisa podatkovnega dela seta, ki se prenese naenkrat. Začetni biti, parni bit in končni biti niso upoštevani. Število podatkovnih bitov je načeloma omejeno od 5 do 9 bitov. Daleč najpogosteje se uporablja dolžina osmih bitov za predstavitev podatka, saj sovpada z dolžino bajta in je delitelj dolžine velike večine podatkovnih vrst. Poleg tega so tudi pomnilniške besede skoraj brez izjeme dolžine, ki je večkratnik števila 8. Na starejših sistemih srečamo tudi uporabo 5 al 7 podatkovnih bitov.

3.2.4 Začetnih bit (Start bit)

Pri vsakem setu, ki se prenese naenkrat, je prvi bit začetni bit, ki signalizira zečetek seta. V serijski komunikaciji se vedno uporablja en začetni bit. Začetni bit pomeni tranzicijo iz negativne napetosti na pozitivno napetost na signalnemu kanalu. Začetni bit je fiksen in v večini primerov ni konfigurabilen.

3.2.5 Končni biti (Stop bits)

Končni biti signalizirajo konec prenešenega seta. V serijski komunikaciji se uporablja eden ali dva takšna bita. Število končnih bitov je večinoma konfigurabilno, vendar nujno 1 ali 2.

3.2.6 Časovna omejitev (Timeout)

Čas v milisekundah. Informacija o tem, kako dolgo naj ukaz "read"v serijski komunikaciji čaka na signal, preden vrne informacijo o napaki.

3.2.7 Strojni nadzor pretoka (Hardware flow control)

Strojni nazdor pretoka je proces kontroliranja hitrosti pretoka glede na zmogljivost procesiranja signala na prejemnikovi strani. Namen nadzora pretoka je preprečitev prehitrega pošiljanja signalov. V kolikor prejemnik ne more pravočasno sprejeti in procesirati vseh poslanih podatkov, lahko pride do zastoja. To se lahko zgodi v primerih, ko se prejemnik sooča z večjo prometno obremenitvijo v primerjavi s pošiljateljem, ali v primerih polnega pomnilniškega prostora na prejemnikovi strani. Strojna rešitev je realizirana z uporabo RTS (Request To Send) in CTS (Clear To Send) signalov, ki potujeta po ločenih kanalih. Takšnemu načinu signalizacija pravimo out-of-band signalizacija. Uporaba ločenih kanalov predstavlja hitrejšo in bolj učinkovito rešitev, vendar pa se tu število kanalov, potrebnih za prenos enega signala, poveča. Nekatere periferne naprave uporabljajo nadzor pretoka, druge te funkcije ne podpirajo.

3.2.8 Programski nadzor pretoka (Software flow control)

Programski nadzor pretoka opravlja isto funkcijo, kot jo izvršuje strojni nadzor, vendar je rešitev realizirana programsko. Pri programskem nadzoru in kontroli pretoka se uporabljata kodi za vklop in izklop prenosa, ki ju označujemo z XON in XOFF. V nasprotju s strojno rešitvijo, se kodi prenašata po istih kanalih, torej kot del sporočila, ki ima posebno obliko. Takšnemu načinu pošiljanja meta podatkov pravimo in-band signalizacija.

Poglavje 4

Generičen serijski gonilnik

Namen tega poglavja je predstaviti in opisati implementacijo generičnega serijskega gonilnika, ki je bil razvit v sklopu diplomske naloge.

Generičen serijski gonilnik definiramo kot gonilnik, ki pri pravilni konfiguraciji opravlja vse naloge specifičnega gonilnika za poljubno napravo. Generičen gonilnik, razvit v sklopu diplomske naloge se poslužuje generičnih funkcij in programskega generiranja programske kode, da na podlagi priložene konfiguracije ustvari specifičen gonilnik za določeno napravo. Zaradi načina delovanja bi lahko torej razvit generičen gonilnik definirali tudi kot aplikacijo za avtomatsko generiranje serijskih gonilnikov.

Razviti generični gonilnik je ustvarjen za okolje Linux. V implementaciji sta uporabljena programska jezika $C++$ in *Python*. Natančneje, programsko generiranje kode in sestavljanje končnega specifičnega serijskega gonilnika je implementirano v programskem jeziku *Python*. Vsak končni serijski gonilnik, ki ga generični ustvari, pa je implementiran v jeziku $C++$. Dodatne razvite knjižnice, ki so skupne vsem generiranim specifičnim gonilnikom, so prav tako implementirane v jeziku $C++$. Generične funkcije so del vsakega generiranega specifičnega gonilnika in so zato tudi implementirane v jeziku $C++$.

Generičen serijski gonilnik je v grobem sestavljen iz dveh glavnih delov, komunikacijskega modula in generičenega modula. Dodatno smo razvili še

tretji, sestavljalni modul, ki olajša implementacijo in delovanje generičnega dela.

Komunikacijski modul skrbi za dejansko komunikacijo. Skrbi za kakovost prenosa in varnost pri samem prenosu podatkov. Zadeva vso konfiguracijo komunikacije, ki je opisana v prejšnjih poglavjih in realizacijo samega pošiljanja signalov preko serijskih vrat. Komunikacijski modul je dokončno implementiran in je kot knjižnica skupen vsem generiranim specifičnim gonilnikom.

Sestavljalni modul zadeva razčlenjevanje in sestavljanje ukazov glede na podan format. Ta modul služi kot orodje generičnemu modulu. Sestavljalni modul je dokončno implementiran in je kot knjižnica skupen vsem generiranim specifičnim gonilnikom.

Generični modul temelji na izdelani osnovi implementacije specifičnega serijskega gonilnika, imenovani *predlogaimplementacijegonilnika*. Ta sestavlja osnovno obliko razreda specifičnega gonilnika, osnovno implementacijo razreda napake in razširitev razreda za sestavljanje ukazov. Razred gonilnika se poslužuje generičnih ukazov za nalaganje parametrov, pošiljanje sporočila in prejemanje odgovora. Drugi del generičnega modula sestavlja branje konfiguracijske datoteke in generiranje vse potrebne programske kode, kot so deklaracije in definicije funkcij, klici nalaganja formatov željenih ukazov, itd. Tretji del zadeva sestavljanje končnega specifičnega gonilnika iz predloge implementacije in generirane programske kode.

Končni specifični gonilnik je razredni objekt tipa singleton. Izpostavlja nabor funkcij, ki pokrivajo celotno funkcionalnost določene naprave. Ves nabor funcij mora biti predhodno specificiran v konfiguracijski datoteki, da jih generičen gonilnik lahko pravilno tvori. Vsaka od funkcij v končni obliki pošlje specifično kodo preko serijskih vrat, ki ji po možnosti sledi ena ali več vrednosti. Koda označuje identiteto ukaza. Koda ukaza in možni parametri funkcije morajo biti določeni v konfiguracijski datoteki. Naprava na nasprotni strani komunikacijskega kanala prepozna identiteto ukaza, prebere možne vrednosti, ki sovpadajo z ukazom, ustrezno reagira in vrne odgovor. V kolikor naprava za določen ukaz ne definira odgovora, pošlje prazno sporočilo.

Prazno sporočilo sestavlja en sam znak, ki je prisoten na koncu vsakega odgovora naprave. Temu znaku pravimo carriage return. Prazen odgovor sporoča, da je naprava prejela ukaz. Funkcija gonilnika v vsakem primeru počaka na odgovor naprave, četudi je ta simboličen, in s tem zagotovi pravilno delovanje. Format odgovora naprave mora biti prav tako določen v konfiguraciji. Funkcija končnega specifičnega gonilnika lahko vrne eno od prejetih vrednosti, ki jih je v odgovor poslala naprava, ali pa ne vrne ničesar. Odgovor funcije mora biti specificiran v konfiguraciji. Pri implementaciji generičnega serijskega gonilnika smo se omejili na sinhroni način izvedbe. To pomeni, da vsaka definirana funkcija počaka na odgovor naprave. Funkcija se vrne šele, ko ga prejme, ali po preteku določenega časa, kar pomeni napako pri komunikaciji.

V sledečih poglavjih so moduli natančneje razčlenjeni in pojasnjeni. Opisan je tudi postopek uporabe gonilnika.

4.1 Komunikacijski modul

Za serijsko komunikacijo in delo s serijskimi vrati smo v implementaciji generičnega gonilnika uporabili standardne UNIX operacije in standardno UNIX knjižnico *termios*. Za odpiranje in zapiranje komunikacije smo se poslužili UNIX operacij *open* in *close*. Za vso konfiguracijo komunikacije se uporablja *termios* knjižnica. Dejansko pisanje in branje skozi serijski vmesnik pa je realizirano s standardnima UNIX operacijama *read* in *write*. Delo s serijskimi vrati je v UNIX sistemih realizirano kot delo z datoteko. Vsakim serijskim vratom pripada določena datoteka, locirana v namenskem direktoriju. Standardna imena teh datotek so oblike *ttyX* (*tty*0, *tty*1, ...), ki so običajno locirani v direktoriju */dev*. Operacije zapiranja, odpiranja, branja in pisanja skozi serijska vrata so za uporabnika identična operacijam nad katerokoli drugo datoteko.

4.1.1 Standardne UNIX operacije

Za delo z datoteko serijskih vrat, uporabljamo standardne POSIX operacije. Te v programski jezik $C++$ vključimo v okviru *unistd* knjižnice.

Za odpiranje datoteke, ki pripada serijskim vratom, smo v implementaciji uporabili standardno UNIX operacijo *open*. Operacija je oblike:

```
int open(const char* filepath, int flags)
```

Prvi vhodni parameter $filepath$ označuje lokacijo datoteke. Pri serijski komunikaciji je najbolj standarden primer tega parametra enak $/dev/tty0$. Drugi vhodni parameter $flags$ določa nastavitve pri odpiranju datoteke. Parameter mora vključevati eno izmed načinov dostopa: O_RDONLY, ki dovoljuje samo branje iz datoteke; O_WRONLY, ki dovoljuje samo pisanje v datoteko; ali O_RDWR, ki dovoljuje oboje. Vključuje lahko tudi druge dodatne nastavitve. Funkcija vrne datotečni opisnik, ki se nadalje uporablja za določanje datoteke pri ostalih operacijah. V primeru napake, funkcija vrne -1 in primerno nastavi parameter *errno*, ki odraža vrsto napake.

Za zapiranje datoteke, ki pripada serijskim vratom, smo v implementaciji uporabili standardno UNIX operacijo *close*. Operacija je oblike:

```
int close(int fd)
```

Prvi vhodni parameter fd je datotečni opisnik, ki označuje datoteko, ki jo zapiramo. Funkcija vrne 0, če je bila datoteka uspešno zaprta. V primeru napake, funkcija vrne -1. V slednjem primeru, funkcija nastavi tudi *errno*, ki označuje vrsto napake.

Za branje podatkov na serijskih vratih smo v implementaciji uporabili standardno UNIX operacijo *read*. Ta bere iz dane datoteke in prebrano prenese v uporabniški buffer. Operacija je oblike:

```
ssize_t read(int fd, void* buffer, size_t count);
```

Prvi vhodni element fd je datotečni opisnik. To je parameter, ki ga vrne UNIX operacija *open* pri odpiranju datoteke. Drugi parameter $buffer$ je kazalec na pomnilniško lokacijo, kamor bo prebrano sporočilo shranjeno. Tretji

parameter *count* označuje pričakovano število bajtov sporočila. Operacija poskuša prebrati do *count* bajtov podatkov iz datoteke, ki jo označuje opisnik fd, in te zapisati na pomnilniško lokacijo z naslovom $buffer$. Pri uspešnem branju, funkcija vrne število prebranih bajtov podatkov. V primeru, da je to enako 0, to označuje konec datoteke oziroma pove, da datoteka ne drži podatkov. Če operacija prebere manjše število bajtov podatkov kakor je bilo to določeno z vhodnim parametrom *count*, to ne označuje napake pri branju. Pomeni le, da željena količina podatkov ni bila na voljo. Po izvršeni operaciji se pri uspešnem branju pomnilniška lokacija datoteke poveča za število bajtov, ki so bili prebrani. To v splošnem pomeni, da se podatki pri branju iz datoteke izbrišejo. Če pride do napake pri branju, funkcija vrne -1. V tem primeru operacija nastavi tudi *errno*, da ta identificira vrsto napake.

Za pisanje oziroma pošiljanje sporočila skozi serijska vrata smo v implementaciji uporabili standardno UNIX operacijo *write*, ki služi za pisanje v dano datoteko. Operacija je oblike:

```
ssize_t write(int fd, const void* buffer, size_t count)
```

Prvi vhodni element fd je datotečni opisnik. Drugi parameter $buffer$ je kazalec na pomnilniško lokacijo, kjer je shranjeno sporočilo. Tretji parameter *count* označuje število bajtov sporočila. Operacija poskuša zapisati *count* bajtov iz lokacije $buffer$ v datoteko, ki jo označuje opisnik fd. Funkcija vrne vrednost, ki označuje uspešnost operacije. V primeru uspešnega pisanja, funkcija vrne število zapisanih bajtov. To število ni nujno enako parametru *count*, saj v nekaterih primerih operacija ne uspe zapisati celotnega sporočila. V slednjem primeru je potrebno preostanek sporočila ponovno pisati. Če funkcija vrne 0, do pisanja ni prišlo. V kolikor funkcija vrne -1, pa je prišlo do napake pri pisanju. V tem primeru operacija nastavi *errno*, da ta identificira vrsto napake.

4.1.2 Termios UNIX knjižnica

Za delo s serijskimi vrati smo v naši implementaciji uporabili standarden UNIX API *termios*, ki dela s terminalnim vhodom/izhodom. Termios definira in uporablja svojo strukturno podatkovno vrsto *struct termios*. Nastavitve serijskih vrat so deljene na pet delov. Termios v svoji strukturni vrsti vsaki namenja svoj atribut.

Tabela 4.1: Nastavitve serijskih vrat z Termios

Vrsta nastavitev	termios atribut
Nastavitve vhoda	tcflag_t c_iflag
Nastavitve izhoda	tcflag_t c_oflag
Nastavitve kontrole	tcflag_t c_cflag
Lokalne nastavitve	tcflag_t c_lflag
Nastavitve posebnih znakov	cc_t c_cc[NCCS]

Termios knjižnica ponuja operacije za branje nastavitev in konfiguracijo komunikacije:

- `int tcgetattr(int fd, struct termios* tio)`

  Nastavitve se preberejo v spremenljivko *tio*, katere naslov je podan kot vhodni parameter operacije.

- `int tcsetattr(int fd, int optional_actions, const struct termios* tio)`

  Parameter *tio*, katerega naslov je podan kot vhodni parameter operacije, določa nove nastavitve komunikacije.

  Parameter *optional_actions* določa, kdaj naj spremembe stopijo v veljavo. Možne nastavitve parametra:

  - $TCSANOW$
  - $TCSADRAIN$

– $TCSAFLUSH$

$TCSANOW$ označuje, naj se spremembe nemudoma uporabijo. $TCSADRAIN$ in $TCSAFLUSH$ določata, naj se spremembe uporabijo šele, ko so vsi podatki, napisani na izhodu vrat, uspešno poslani. $TCSAFLUSH$ poleg tega vse podatke, prejete preko serijskih vrat, ki še niso bili prebrani, pred uporabo novih nastavitev zavrže.

Poseben parameter je stopnja Bauda. Parameter je del *struct termios* vrste, vendar se za branje in pisanje stopnje Bauda v *structtermios* objekt uporabljajo posebni ukazi:

- `speed_t cfgetispeed(const struct termios *tio)`
- `speed_t cfgetospeed(const struct termios *tio)`
- `int cfsetispeed(struct termios *tio, speed_t speed)`
- `int cfsetospeed(struct termios *tio, speed_t speed)`

Vrsta *speed_t* določa stopnjo Bauda, ki mora biti ena iz seta dovoljenih konstant, navedenih v tabeli 4.2. Za konfiguracijo stopnje Bauda je po izvršitvi $cfsetispeed$ ali $cfsetospeed$ potrebno izvšiti še operacijo $tcsetattr$.

Tabela 4.2: Stopnje Bauda

B0	B50	B75	B110	B134
B150	B200	B300	B600	B1200
B1800	B2400	B4800	B9600	B19200
B38400	B57600	B115200	B230400	

Pri branju s serijskih vrat se poraja vprašanje, kako dolgo, če sploh, čakati na podatke in koliko od teh zadostuje za branje. Kot rešitev knjižnica *termios* skozi atribut *c_cc* podaja parametra $VMIN$ in $VTIME$. Poleg tega se pri vsakem branju posebej poda parameter $NBYTES$, ki določa število znakov, ki jih pričakujemo. $VMIN$ označuje minimalno število znakov, ki

se morajo prebrati. $VTIME$ označuje časovno omejitev čakanja na podatke pri branju. Ločimo štiri različne kombinacije, ki se razlikujejo po delovanju:

- $VMIN = 0$ and $VTIME = 0$

  Ta način označuje popolnoma neblokirajoče branje. Če so podatki prisotni, se jih do $NBYTES$ prenese v klicateljev buffer. Če podatki niso na voljo, branje vrne 0.

- $VMIN = 0$ and $VTIME > 0$

  Ta način označuje branje s čisto časovno omejitvijo. Če so podatki prisotni, se jih do $NBYTES$ prenese v klicateljev buffer. Če podatki niso na voljo, branje čaka, dokler ti ne prispejo, ali dokler se časovno omejitev ne izteče. Če podatkov po preteku časovne omejitve ni, branje vrne 0. Za izpolnitev branja zadostuje že en sam znak, vendar pa se jih prenese do $NBYTES$, kolikor jih je na voljo. Časovno omejitev v tem primeru merimo od začetka operacije, do konca operacije.

- $VMIN > 0$ and $VTIME > 0$

  Operacija branje se vrne, ko je na voljo vsaj $VMIN$ znakov, ali ko preteče $VTIME$ čas med dvema prebranima znakoma. Tu se časovna omejitev nanaša na čas med enim in drugim prebranim znakom, in ne na celoten pretečen čas. Ker se čas začne meriti šele s prvim prispelim znakom, lahko operacija branja blokira za nedoločen čas, če serijska komunikacija miruje. Branje v tem načinu nikoli ne vrne 0.

- $VMIN > 0$ and $VTIME = 0$

  Ta način označuje čisto števno branje. Operacija branje se vrne, ko je na voljo vsaj $VMIN$ znakov. Na klicateljev buffer prenese do $NBYTES$ znakov, kolikor jih je na voljo v trenutku, ko je $VMIN$ pogoj zadoščen. Operacija branja lahko blokira za nedoločen čas.

4.1.3 Predstavitev modula

Konfiguracijski modul predstavlja semi-singleton razred *Komunikator*. Instanca razreda *Komunikator* navzven ponuja nabor funkcij za kreiranje in uničenje instance razreda, konfiguracijo, povezovanje s serijskimi vrati, funkcije za pošiljanje in prejemanje niza podatkov skozi serijska vrata in funkcije za izplakovanje komunikacijskega kanala. Natančneje, implementirali smo naslednje javne funkcije:

- `static SerialComm* getInstance(const char* portFile)`

  Funkcija za pridobitev instance razreda *Komunikator*. Argument *portFile* predstavlja lokacijo datoteke na disku, ki pripada enim od serijskih vrat. Za vsako od teh dopustimo samo eno instanco razreda *Komunikator*. V kolikor pride do zahteve po instanci razreda za ista serijska vrata večkrat, funkcija vedno vrne isto instanco. Pri zahtevi po instanci razreda za še ne zahtevana serijska vrata, funkcija kreira novo instanco razreda, jo shrani in vrne. Pri kreiranju nove instance, ta med svojo konstrukcijo izvrši lastno funkcijo *connect* za povezavo s serijskimi vrati in samodejno nastavi privzete konfiguracijske nastavitve komunikacije.

- `virtual ~SerialComm()`

  Funkcija, ki se izvrši ob uničenju instance razreda. Funkcija poskuša izvršiti lastno funckijo *disconnect*, ki poskrbi za odklop povezave in zapre serijska vrata.

- `void connect()`

  Funkcija ki vpostavi povezavo s serijskimi vrati in nastavi osnovne privzete nastavitve parametrov serijskih vrat.

  Serijska vrata odpre s klicem:

```
int fd;
fd = open(portFile, O_RDWR | O_NOCTTY | O_NONBLOCK);
```

Argument *portFile* predstavlja lokacijo datoteke na disku, ki pripada serijskim vratom. *O_RDWR* označuje bralni in pisalni dostop do vrat. *O_NOCTTY* določuje, da če *portFile* označuje terminalno napravo, operacija *open* ne bo povzročila, da bi ta postala kontrolni terminal procesa. *O_NONBLOCK* določa, da se bo operacija *open* vrnila brez čakanja na potrditev naprave na nasprotni strani, da je ta pripravljena in dosegljiva. V kolikor operacija *open* vrne < 0, pomeni, da je prišlo do napake pri odpiranju serijskih vrat. *Komunikator* v tem primeru izstavi izjemo.

Argumente, povezane s terminalom pridobi z naslednjim klicem:

```
struct termios tio;
tcgetattr(fd, &tio);
```

V kolikor operacija *tcgetattr* vrne < 0, je prišlo do napake pri branju atributov. *Komunikator* v tem primeru izstavi izjemo.

Funkcija nastavi naslednje nastavitve:

```
tio.c_iflag = IGNBRK | IGNPAR;
tio.c_oflag = 0;
tio.c_lflag = 0;
tio.c_cc[VMIN] = 0;
tio.c_cc[VTIME] = 0;
```

Z uporabo *IGNBRK* ignoriramo stanje BREAK na vhodu. Z *IGNPAR* ignoriramo napake seta in napake pri parnosti.

Funkcija nastavi vrednosti s klicem:

```
tcsetattr(fd, TCSANOW, &tio)
```

TCSANOW označuje, naj se spremembe nemudoma uporabijo.

- `void disconnect()`

Funkcija, ki odklopi povezavo in zapre serijska vrata. Funkcija najprej preveri, ali so serijska vrata dotične instance komunikacijskega razreda odprta. V kolikor so, te zapre s klicem:

```
ret = close(fd);
fd = -1;
```

Operacija *close* pri pravilnem delovanju vrne 0. V nasprotnem primeru je prišlo do napake pri zapiranju vrat. *Komunikator* v tem primeru izstavi izjemo.

- `void setTimeout(float timeout)`

  Funkcija ki nastavi čas, podan v sekundah, kot čas, ki je dovoljen za čakanje na podatke za branje preko serijskih vrat. Po pretečenem času se branje smatra kot neuspešno.

  Funcija nastavi časovno omejitev z uporabo sledečih klicev:

  ```
  tio.c_cc[VMIN] = 0;
  tio.c_cc[VTIME] = TIMEOUT_IN_DSEC;
  tcsetattr(fd, TCSANOW, &tio)
  ```

  $TIMEOUT\_IN\_DSEC$ označuje podano časovno omejitev, preračunano v decisekunde. V primeru, ko je podana časovna omejitev manjša od 0, torej neveljavna, funkcija nastavi popolen čakajoči način. $VMIN$ nastavi na 1, $VTIME$ nastavi na 0.

- `void setEos(string eos)`

  Funcija, ki nastavi EOS(angl. end of string) znak. EOS znak označuje konec niza. Branje se pri prejetju tega znaka ustavi in prenese prebrano. Pri serijski komunikaciji se v večini primerov kot EOS uporablja CR(angl. carriage return), ki določa konec poslanega seta.

- `void setBaud(int baud)`

  Funkcija, ki nastavi stopnjo Bauda.

Stopnjo Bauda nastavi z uporabo sledečih klicev:

```
cfsetispeed(&tio,BAUD_CODE)
cfsetospeed(&tio,BAUD_CODE)
tcsetattr(fd, TCSANOW, &tio)
```

Parameter $BAUD\_CODE$ določa stopnjo Bauda. Vhodno in izhodno hitrost nastavimo na željeno stopnjo. Če katerikoli izmed navedenih klicev vrne < 0, potem je prišlo do napake pri konfiguraciji stopnje Bauda. V tem primeru *Komunikator* izstavi izjemo.

- `int getBaud()`

  Funkcija, ki vrne stopnjo Bauda.

  Stopnjo Bauda prebere z uporabo sledečih operacij:

  ```
  iBaud = cfgetispeed(&tio)
  oBaud = cfgetospeed(&tio)
  ```

  V implementaciji razreda *Komunikator* smo privzeli, da sta vhodna in izhodnja stopnja enaki. V primeru, da pri branju stopnji nista enaki, *Komunikator* izstavi izjemo. V nasprotnem primeru vrne vrednost.

- `void setDataBits(short bits)`

  Funkcija, ki nastavi število podatkovnih bitov.

  To doseže z operacijami:

  ```
  tio.c_cflag = (tio.c_cflag & ~CSIZE) | CS_NUM
  tcsetattr(fd, TCSANOW, &tio)
  ```

  Parameter CS_NUM je enak eni izmed konstant, ki označujejo število podatkovnih bitov. Število teh mora biti med 5 in 8. V primeru, da je vhodni parameter *bits* enak 5, 6, 7 ali 8, je CS_NUM enak CS_5, CS_6, CS_6 ali CS_8, zaporedno pripadajoče. V kateremkoli drugem primeru, *Komunikator* izstavi izjemo.

- `short getDataBits()`

  Funkcija, ki vrne število podatkovnih bitov v uporabi.

  To doseže z operacijami:

  ```
  CS_NUM = tio.c_cflag & CSIZE
  ```

  V primeru, da je CS_NUM enak CS_5, CS_6, CS_7 ali CS_8, funkcija vrne 5, 6, 7 ali 8.

- `void setStopBits(short bits)`

  Funkcija, ki nastavi število končnih bitov.

  V primeru, da je vhodni parameter *bits* enak 1, funckija nastavi:

  ```
  tio.c_cflag &= ~CSTOPB
  ```

  V primeru, da je vhodni parameter enak 2:

  ```
  tio.c_cflag |= CSTOPB
  ```

  V ostalih primerih *Komunikator* izstavi izjemo.
  Spremembo uveljavi s klicem:

  ```
  tcsetattr(fd, TCSANOW, &tio)
  ```

- `short getStopBits()`

  Funkcija, ki vrne število končnih bitov v uporabi. To število je lahko 1 ali 2.

  Število končnih bitov dobimo z operacijo:

  ```
  bits = (tio.c_cflag & CSTOPB) ? 2 : 1
  ```

- `void setParity(short parity)`

  Funkcija, ki nastavi pravilo parnega bita.

  V primeru, da je vhodni parameter *parity* enak 0, funkcija odstrani uporabo parnega bita:

```
tio.c_cflag &= ~PARENB
```

V primeru, da je vhodni parameter enak 1, funkcija nastavi uporabo lihega parnega bita:

```
tio.c_cflag |= PARENB
tio.c_cflag |= PARODD
```

V primeru, da je vhodni parameter enak 2, funkcija nastavi uporabo sodega parnega bita:

```
tio.c_cflag |= PARENB
tio.c_cflag &= ~PARODD
```

V ostalih primerih *Komunikator* izstavi izjemo.
Spremembo uveljavi s klicem:

```
tcsetattr(fd, TCSANOW, &tio)
```

- `short getParity()`

  Funkcija, ki vrne pravilo parnega bita. V primeru onemogočenega parnega bita, funkcija vrne 0. Pri uporabi lihega parnega bita, funkcija vrne 1. Pri uporabi sodega parnega bita, funkcija vrne 2.

  Uporabo parnega bita preveri s klicem:

  ```
  if(tio.c_cflag & PARENB)
  ```

  Lihost pravila parnega bita ugotovi s klicem:

  ```
  if(tio.c_cflag & PARODD)
  ```

- `void setClocal(bool clocal)`

  Funkcija, ki nastavi parameter $CLOCAL$. Ta nastavitev se upošteva samo pri komunikaciji z modemsko vodeno napravo. Pri uporabi te nastavitve, ignoriramo statusne kanale naprave. V nasprotnem primeru so te nadzorovane.

  Ko je vhodni parameter *clocal* enak $True$, funkcija nastavi parameter:

```
tio.c_cflag |= CLOCAL
```

Ko je ta enak $False$, funkcija odstrani nastavitev:

```
tio.c_cflag &= ~CLOCAL
```

Spremembo nastavitve uveljavi s klicem:

```
tcsetattr(fd, TCSANOW, &tio)
```

- `bool getClocal()`

  Funkcija, ki vrne nastavitev parametra $CLOCAL$.

  Nastavitev pridobi s klicem:

  ```
  CLOCAL = (tio.c_cflag & CLOCAL) ? True : False
  ```

- `void setHardwareFlowControl(bool crtscts)`

  Funkcija, ki omogoči ali onemogoči uporabo strojnega nadzora pretoka.

  Ko je vhodni parameter *crtscts* enak $True$, funkcija omogoči strojni nadzor pretoka:

  ```
  tio.c_cflag |= CRTSCTS
  ```

  Ko je ta enak $False$, funkcija onemogoči strojni nadzor pretoka:

  ```
  tio.c_cflag &= ~CRTSCTS;
  ```

  Spremembo nastavitve uveljavi s klicem:

  ```
  tcsetattr(fd, TCSANOW, &tio)
  ```

- `bool getHardwareFlowControl()`

  Funkcija, ki pove, ali je strojni nadzor pretoka omogočen.

  Nastavitev pridobi s klicem:

  ```
  HFC = (tio.c_cflag & CRTSCTS) ? True : False
  ```

- `void setSoftwareFlowControl(bool sfc)`

  Funkcija, ki omogoči ali onemogoči uporabo programskega nadzora pretoka.

  Ko je vhodni parameter sfc enak $True$, funkcija omogoči programski nadzor pretoka:

  ```
  tio.c_iflag |= (IXON | IXOFF | IXANY);
  ```

  Ko je ta enak $False$, funkcija onemogoči programski nadzor pretoka:

  ```
  tio.c_iflag &= ~(IXON | IXOFF | IXANY);
  ```

  Spremembo nastavitve uveljavi s klicem:

  ```
  tcsetattr(fd, TCSANOW, &tio)
  ```

- `bool getSoftwareFlowControl()`

  Funkcija, ki pove, ali je programski nadzor pretoka omogočen.

  Nastavitev pridobi s klicem:

  ```
  SFC = (tio.c_cflag & (IXON | IXOFF | IXANY)) ? True : False
  ```

- `void serialWrite(char* sendBuffer, int sendBufferLength)`

  Funkcija, ki skozi serijska vrata pošlje sporočilo oziroma ukaz. Sporočilo je podano z vhodnim parametrom $sendBuffer$. Dolžina sporočila oziroma število znakov sporočila je podano z vhodnim parametrom $sendBufferLength$. Funkcija najprej preveri, ali je serijska povezava vzpostavljena. V primeru da ni, povezavo vzpostavi s klicem funkcije *connect*. Nato sledi dejansko pisanje. Pisanje se izvaja z operacijo *write*.

  Psevdokoda implementacije funkcije:

  ```
  1. Če povezava ni vpostavljena:
      connect()
  2. Izmeri trenutni, začetni čas
  ```

```
3. Neskončna zanka:
    3.1 št_zapisanih = write(fd, buffer, št_preostalih)
    3.2 Če št_zapisanih >= 0:
        3.2.1 št_preostalih = št_preostalih - št_zapisanih
        3.2.2 Če št_preostalih == 0:
            3.2.2.1 Konec
        3.2.3 Zamakni buffer kazalec za št_zapisanih naprej
    3.3 Če število zapisanih < 0:
        3.3.1 Če napaka ni v: [EWOULDBLOCK, EINTR, EAGAIN]:
            3.3.1.1 disconnect()
            3.3.1.2 Izstavi izjemo
    3.4 Izmeri trenutni čas in zračunaj pretečen čas
    3.5 Če je časovna omejitev prekoračena:
        3.5.1 Izstavi izjemo
```

- `void serialRead(char* responseBuffer, int responseBufferLength)`

  Funkcija, ki iz datoteke serijskih vrat prebere prispele podatke. Funkcija poskuša prebrati toliko bajtov podatkov, kolikor veleva drugi vhodni parameter $responseBufferLength$. Podatke shrani na pomnilnik na naslov, ki ga določa kazalec $responseBuffer$. Funkcija najprej preveri, ali je serijska povezava vzpostavljena. V primeru da ni, povezavo vzpostavi s klicem funkcije *connect*. Nato sledi dejansko branje iz datoteke serijskih vrat. To se izvaja z operacijo *read*.

  Psevdokoda implementacije funkcije:

```
1. Če povezava ni vpostavljena:
    connect()
2. Neskončna zanka:
    2.1 št_prebranih = read(fd, buffer, št_preostalih)
    2.2 Če št_prebranih > 0:
        2.2.1 Če je parameter $eos$ nastavljen:
            2.2.1.1 Če je $eos$ prisoten v prebranih podatkih:
                2.2.1.1.1 Za $eos$ v bufferju zapiši 0
```

```
                2.2.1.1.2 Konec
        2.2.2 Zamakni buffer kazalec za št_prebranih naprej
        2.2.3 št_preostalih = št_preostalih - št_prebranih
        2.2.4 Če št_preostalih == 0:
            3.2.2.1 Konec
    2.3 Če št_prebranih == 0:
        2.3.1 Izstavi izjemo
    2.4 Če št_prebranih < 0:
        2.4.1 Če napaka ni v: [EWOULDBLOCK, EINTR, EAGAIN]:
            2.4.1.1 disconnect()
            2.4.1.2 Izstavi izjemo
```

- `void flush()`

  Funkcija, ki spere podatke iz datoteke serijskih vrat. Funkcija se uporablja v primerih nerešljive napake v komunikaciji in poskrbi za novo prazno osnovo.

  Psevdokoda implementacije funkcije:

```
1. Če povezava ni vpostavljena:
    connect()
2. Neskončna zanka
    2.1 status = read(fd, začasni_buffer, 1)
    2.2 Če status <= 0:
        2.2.1 Konec
```

4.2 Sestavljalni modul

Sestavljalni modul zajema funkcionalnost za razčlenjevanje in sestavljanje ukazov glede na podan format. Služi kot orodje generičnemu modulu, opisanemu v sledečem poglavju.

Sestavljalni modul implementira razred *Sestavljalnik*. Poleg tega modul zajema tudi lastno verzijo razreda izjeme *SestavljalnikException*. Razred

Sestavljalnik ponuja funkcionalnost za podajanje formata ukaza in odgovora operacije. Pri podanih formatih ukazov, *Sestavljalnik* ponuja funkcije za sestavljanje ukaznega niza znakov iz dejanskih podanih vrednosti, glede na prej specificiran format. Generičen modul uporablja to funkcionalnost za sestavljanje paketa, ki ga z uporabo komunikacijskega module pošlje napravi. *Sestavljalnik* pri podanih formatih odgovorov ponuja funkcijo za razčlenjevanje odgovora v niz dejanskih vrednosti. Generičen modul uporablja to funkcionalnost za branje in razčlenjevanje odgovora naprave, ki ga prejme z uporabo komunikacijskega modula. V nadaljevanju sledi obrazložitev delovanja teh treh operacij sestavljalnega modula.

4.2.1 Podajanje formatov

Za vsako operacijo, ki jo lahko zahtevamo od naprave, poznamo format zahtevka, ki ga moramo poslati napravi, in format odgovora, ki ga prejmemo.

Format zahtevka in odgovora za neko operacijo definiramo z uporabo funkcije:

```
void addFormat(const unsigned int operationID,
        string commandF, string responseF)
```

Prvi vhodni parameter *operationID* določa unikaten ID operacije. V primeru, da format z istim ID operacije že obstaja, bo funkcija format spremenila na novo vrednost. Drugi vhodni *commandF* parameter določa format ukaza operacije. Tretji vhodni parameter *responseF* določa format odgovora operacije. Formata morata biti podana na sledeč način:

Tabela 4.3: EBNF format Podajanja formata

FORMAT	{ {ASCII_CHAR \"%"} , {"%%"} , {"%s"} , {"%",X,"s"} }

V stringu, ki podaja format, so lahko prisotni poljubni ASCII znaki, na poljubnih mestih, z izjemo znaka %. Prav tako so lahko prisotni nizi %%, %*s*

in $\%Xs$(npr. %3s), poljubnokrat na poljubnih mestih. *Sestavljalnik* bo format interpretiral sledeče: Za vsak pojav niza $\%s$ ali $\%Xs$, bo ta niz interpretiran kot nosilec za neko vrednosti. Pri sestavljanju ukaza ali razčlenjevanju odgovora bo ta niz zamenjal z dejansko vrednostjo. Pri pojavu $\%Xs$ bo vrednost, ki jo bere ali vstavlja, omejil na X znakov. Pri pojavu niza %%, bo ta zamenjan z nizem %. Ostalih znakov *Sestavljalnik* ne bo zamenjal. V končni fazi, s formatom definiramo zaporedje ASCII znakov, ki sestavljajo ukaz ali odgovor, med katere lahko vmešamo nosilce(angl. placeholders) za dejanske vrednosti, ki pa jih bomo podali pri samem sestavljanju ukaza ali razčlenjevanju odgovora.

4.2.2 Sestavljanje ukaza

Za sestavljanje ukaza *Sestavljalnik* ponuja funkcijo:

```
string composeCommand(const unsigned int operationID,
        vector<string>* values)
```

S prvim vhodnim parametrom *operationID* identificiramo operacijo, katere ukaz sestavljamo. Format ukaza za to operacijo mora biti vnaprej definiran z uporabo zgoraj opisane funkcije. V nasprotnem primeru funkcija izstavi izjemo. Z drugim vhodnim parametrom *values*, funkciji podamo serijo dejanskih vrednosti, ki jih bo ta uporabila za sestavo ukaza. Število podanih vrednosti se mora ujemati s številom nosilcev vrednosti ($\%s$ ali $\%Xs$) v formatu ukaza. V nasprotnem primeru funkcija izstavi izjemo. Elementi serije *values* morajo biti tipa *string*, kar pomeni da morajo biti števila vnaprej pretvorjena v tip *string*. Za to sestavljalni modul ponuja dve dodatni funkciji za pretvarjanje števil v in iz tipa *string*.

Funkcija prebere format ukaza dotične operacije. Vse instance niza %% zamenja z %. Instance niza $\%s$ in $\%Xs$ po vrsti zamenja z vrednostmi iz parametra *values*, kjer se pri nizu $\%Xs$ omeji na X znakov vrednosti. Nastal niz znakov funkcija vrne v odgovor.

4.2.3 Razčlenjevanje odgovora

Za razčlenjevanje odgovora *Sestavljalnik* ponuja funkcijo:

```
void parseResponse(const unsigned int operationID,
        string response, vector<string>* values)
```

S prvim vhodnim parametrom *operationID* identificiramo operacijo, katere odgovor razčlenjamo. Format odgovora za to operacijo mora biti vnaprej definiran z uporabo prej opisane funkcije. V nasprotnem primeru funkcija izstavi izjemo. Z drugim vhodnim parametrom *response* funkciji podamo odgovor naprave oziroma string, ki ga bo *Sestavljalnik* razčlenil. S tretjim vhodnim parametrom *values* funkciji podamo prazno serijo vrednosti, ki jo funkcija napolni z dejanskimi vrednostmi.

Funkcija prebere format odgovora dotične operacije in podan string *response*. Oba stringa se morata ujemati, upoštevajoč, da kjer v formatu piše %%, mora v *response* pisati %; kjer v formatu piše $\%s$, lahko v *reponse* na tem mestu leži poljuben string; in kjer v formatu piše $\%Xs$, mora v *response* na tem mestu ležati poljuben string dolžine X. Vse stringe, ki v *response* nadomeščajo $\%s$ ali $\%Xs$, funkcija izlušči in prepiše v serijo stringov *values*, po vrsti. V kolikor se stringa ne ujemata po zgoraj opisanih pravilih, funkcija izstavi izjemo.

4.2.4 Primeri

Za demonstracijo funkcije *composeCommand*, predpostavimo naslednje vrednosti:

- Format ukaza: A1%s%s%%
- Vrednosti: 1.2, 3.4

Odgovor funkcije *composeCommand* bo enak:

```
A11.23.4%
```

Za demonstracijo funkcije *parseResponse*, predpostavimo naslednje vrednosti:

- Format odgovora: B2%%%3sX%sY
- Odgovor: B2%8.8X9.9Y

Vrednosti po klicu funkcije *parseResponse* bodo:

```
[8.8, 9.9]
```

4.3 Generičen modul

Generičen modul zajema predlogo implementacije, branje konfiguracije, celotno generično funkcionalnost in končno programsko generiranje programske kode. Del generičnega modula, v sklopu predloge implementacije, sestavlja okostje razreda končnega gonilnika in vseh pomožnih razredov, na podlagi katerih, po navodilih konfiguracije, generičen modul gradi. Generičen modul je osrednji del generičnega gonilnika. Predhodna modula služita kot orodje generičnemu modulu.

Če povzamemo grobo strukturo delovanja generičnega modula. Generičen modul kot osnovo vzame predlogo implementacije končnega gonilnika. Strukturo te predloge bomo podrobneje pogledali v naslednjem poglavju. Naj v grobem povemo, da predlogo med drugimi sestavlja struktura razreda končnega gonilnika, z dodanimi tremi generičnimi funkcijami. Generične funkcije služijo za nalaganje podatkov za prenos; za odpošiljanje sporočila glede na podano operacijo in naložene podatke; ter za prejemanje odgovora, ki prav tako sloni na podani operaciji. Omenjene generične funkcije so podrobneje opisane v sledečih poglavjih. Generičen modul v naslednji fazi prebere konfiguracijsko datoteko. Na podlagi te, implementacijo končnega gonilnika dopolni s potrebno programsko kodo. Generirana programska koda se poslužuje generičnih funckij. Velik del funkcinalnosti, tako programsko generirane, kot tiste iz predloge, pa se poslužuje komunikacijskega in sestavljalnega modula, ki smo ju predstavili v prejšnjih poglavjih. Generiranje

programske kode bomo prav tako podrobneje opredelili v enem od sledečih poglavij.

Končni rezultat, ki ga ustvari generičen modul je končni specifični gonilnik. Ta pri delovanju za knjižnico uporablja komunikacijski in sestavljalni modul. Nastali specifični gonilnik navzven ponuja nabor funkcij, ki jih je predhodno generiral generični modul, po specifikacijah konfiguracije. Vsaka funkcija pripada eni operaciji, ki jo končni gonilnik, preko serijske komunikacije, lahko izvrši na priključeni napravi. Vhodni parametri vsake funkcije sovpadajo z vhodnimi parametri operacije naprave. Izhodni parameter vsake funkcije pa sovpada z enim izmed parametrov odgovora naprave. Nabor funkcij in njihove specifikacije generičen modul pričakuje v konfiguracijski datoteki.

4.3.1 Predloga implementacije gonilnika

Predlogo implementacije gonilnika sestavljajo štiri datoteke, ki skupno definirajo štiri razrede. V nasledjih poglavjih bomo spoznali, da generičen modul z generiranjem kode zgolj dopolnjuje definirane razrede. Prav tako ne ustvarja novih datotek, ampak obstoječe datoteke predloge samo kopira in preimenuje. Tako predloga implementacije gonilnika dobro definira končno strukturo generiranih specifičnih gonilnikov.

V predlogi implementacije se pojavljajo nosilci(angl. placeholders). Prvi in zadnji znak vsakega nosilca je enak znaku $, med njima pa je navedena identiteta nosilca. Pri samem generiranju kode, generični modul zamenja vse nosilce z dejansko vrednostjo ali programsko kodo, ki jo opredeljuje identiteta nosilca. Nosilci v predlogi tako označujejo mesta, kjer naj bo vnešena določena generirana koda. Podrobnejša razlaga generiranja programske kode sledi v nasledjih poglavjih, pomeni nosilcev pa so razloženi v tabeli 4.4.

Datoteke predloge:

- _Driver.h
- _Driver.cpp
- _Exception.h
- _Parser.h

Razredi implementacije gonilnika:

- $NAME$Driver
- $NAME$Guard
- $NAME$Exception
- $NAME$Parser

4.3.2 Generične funkcije

Kot smo omenili, za vsako, v konfiguracijski datoteki specificirano operacijo, ki se preko serijske komunikacije lahko zahteva od naprave, generični gonilnik zanjo programsko generira funkcijo v končnemu gonilniku. Vsaka taka funkcija izvaja podobno delo: vhodne parametre, skupaj z identiteto operacije, pošlje preko serijskega kanala. Od naprave kot odgovor prejme sporočilo, ki ga specifično za operacijo pretvori in vrne v obliki izhodnih parametrov.

Za lažjo implementacijo oziroma lažje generiranje funkcij gonilnika in preprečevanje podvajanja kode, se funkcije končnega gonilnika za večino dela poslužujejo generičnih funkcij *Load*, *Send*, *Respond*. Te so že vnaprej implementirane v predlogi implementacije. Njihova generičnost zagotavlja pravilno delovanje na poljubnem številu podatkov pri uporabi poljubnih podatkovnih vrst le teh.

Za vsako vrednost, ki jo v sklopu pripadajoče operacije po serijskem kanalu želi poslati neka funckija končnega gonilnika, ga ta v prvi fazi, z uporabo generične funkcijne *Load*, naloži na vektor vrednosti. Ko funkcija tako naloži vse parametre operacije, z uporabo funkcije *Send* pošlje sporočilo preko serijskega kanala. Sporočilo zajema kodo operacije in vse naložene parametre. V zadnjem koraku funkcija, z uporabo generične funkcije *respond*, pridobi in izlušči iskano vrednost, ki jo naprava pošlje v odgovor.

Omenjene generične funkcije se dodatno poslužujejo še dveh dodatnih generičnih metod: $fromString$ in *toString*. Prva od teh pretvori vrednost tipa *string* v zahtevano poljubno podatkovno vrsto. Druga opravlja nasprotno

Tabela 4.4: Nosilci v predlogi implementacije

Nosilci vrednosti in razlage	
$NAME$	
	Nosilec za ime gonilnika. Ta se pojavlja rezmeroma skozi celotno predlogo implementacije, saj imena vseh razredov v sklopu končnega gonilnika vključujejo to ime.
$NAMEU$	
	Nosilec za ime, zapisano z velikimi tiskanimi črkami. Uprablja se v zapisih vključevalnih varoval (angl. include guards) razredov.
$FUNCTION_DECLARATIONS$	
	Nosilec za deklaracije funkcij. Uporablja se izključno v header datoteki gonilnika, na mestu kjer naj bodo locirane deklaracije funkcij razreda $\$NAME\$Driver$. Vsaka funkcija bo namenjena eni operaciji, ki jo preko serijske komunikacije na priključeni napravi lahko izvši gonilnik.
$FUNCTION_DEFINITIONS$	
	Nosilec za definicije funkcij. Uporablja se izključno v source datoteki gonilnika, na mestu kjer naj bodo locirane definicije funkcij razreda $\$NAME\$Driver$. Vsaka funkcija bo izvršila eno operacijo na priključeni napravi. Vhodni in izhodni parametri funkcije sovpadajo s parametri pripadajoče operacije.
$COMMAND_ENUM$	
	Nosilec za enumerator vseh operacij. Uporablja se v implementaciji razreda $NAME$Parser. Enumerator operacij služi kot niz identitet vseh končnih operacij, ki jih bo zajemal razred $NAME$Driver. Vsaka generirana funkcija končnega gonilnika bo pripadala eni izmed teh.
$COMMANDS_CREATE$	
	Nosilec, kjer se bodo, z uporabo sestavljalnega modula, definirale vse operacije. Za vsako, v konfiguracijski datoteki definirano operacijo, ki se lahko izvrši na priključeni napravi, bo tu na podlagi njene identitete podan format zahtevka in format odgovora le te.

funkcijo.

Generična funkcija Load

Funkcija za nalaganje parametra na vektor vrednosti. Funkcija je oblike:

```
template<class L>
void $NAME$Driver::Load(L load) throw($NAME$Exception)
```

Vhodni parameter je lahko poljubne podatkovne vrste. Funkcija vhodni parameter najprej pretvori v podatkovno vrsto tipa *string*, nakar pretvorjeno vrednost naloži na namenski vektor vrednosti, kjer se hranijo že naložene vrednosti. V primeru napake pri pretvarjanju ali nalaganju, funkcija izstavi izjemo.

Generična funkcija Send

Funkcija za pošiljanje zahtevka po operaciji po serijskem kanalu. Funckija je oblike:

```
void $NAME$Driver::Send(comm_name_t commandID) throw($NAME$Exception)
```

Vhodni parameter *commandID* identificira operacijo. Format zahtevka in format odgovora za dotično operacijo morata biti vnaprej definirana v sestavljalnem modulu z uporabo metode *addFormat*. V nasprotnem primeru funkcija izstavi izjemo.

Funkcija uporabi metodo *composeCommand* sestavljalnega modula, da sestavi besedilo zahtevka:

```
command = commandParser.composeCommand(commandID, &values);
command += 0x0D;
```

Prvi parameter *commandID* pri klicu metode *composeCommand* je enak identiteti operacije, ki jo generična funkcija *Send* prejme kot vhodni parameter. Drugi parameter *values* je vektor vrednosti, ki jih bo metoda uprabila pri sestavljanju zahtevka. Ta vektor je niz vrednosti, ki jih je dotična

funkcija končnega gonilnika pred klicem funkcije *Send* naložila z uporabo generične funkcije *Load*. Kot je podrobneje opisano v prejšnjih poglavjih, metoda *composeCommand* na podlagi predhodno definiranih formatov za določeno operacijo in z uporabo podanih vrednosti sestavi in vrne besedilo zahtevka. Funkcija *Send* dobljenemu zahtevku na koncu doda znak CARRIAGE_RETURN, v šestanjstiškem zapisu je ta enak 0x0D, ki označuje konec sporočila.

V naslednji fazi funkcija *Send* z uporabo komunikacijskega modula pošlje sporočilo po komunikacijskem kanalu:

```
serialComm->serialWrite((char*)(command.c_str()), command.size());
```

Pred vrnitvijo, funckija *Send* pobriše namenski vektor vrednosti, ki se polni z uporabo funkcije *Load*. Omenimo še, da v sklopu funckije *Send* odgovora naprave ne beremo.

Generična funkcija Respond

Funkcija za branje odgovora periferne naprave po izvršeni operaciji. Funkcija je oblike:

```
template <class R>
R $NAME$Driver::Respond(comm_name_t commandID, int responseLength,
                int responseNum) throw($NAME$Exception)
```

Prvi vhodni parameter *commandID* identificira operacijo. Format zahtevka in format odgovora za dotično operacijo morata biti vnaprej definirana v sestavljalnem modulu z uporabo metode *addFormat*. V nasprotnem primeru funkcija izstavi izjemo. Drugi vhodni parameter *responseLength* označuje pričakovano dolžino odgovora v bajtih. Tretji vhodni parameter *responseNum* pa označuje pozicijo elementa v odgovoru naprave, ki naj ga funkcija *Respond* vrne. Najpogosteje naprava v odgovor na neko zahtevo ne pošlje ničesar, oziroma pošlje prazen odgovor, ki oznanja, da je naprava prejela zahtevek. V takem primeru se pričakuje vrednost $responseLength = 1$ in vrednost $responseNum = 0$. V drugih primerih naprava kot odgovor vrne

eno izmerjeno vrednost. V slednjem primeru je vrednost *responseLength* odvisna od tipa vrednosti, vrednost *reponseNum* pa naj bi bila enaka 0. Najmanj pogosto naprava kot odgovor na neko zahtevo vrne več vrednosti. V tem primeru je vrednost *responseLength* odvisna od dolžine vseh prisotnih elementov v odgovoru. Vrednost *reponseNum* pa mora kazati na eno izmed vrednosti. V primeru, da naprava vrne dve vrednosti, je ta lahko enaka 0 ali 1.

Funkcija z uprabo komunikacijskega modula prebere odgovor naprave v interni buffer:

```
serialComm->serialRead(&buffer[0], responseLength);
```

V naslednji fazi, z uporabo sestavljalnega modula, odgovor naprave razdre v namenski vektor vrednosti:

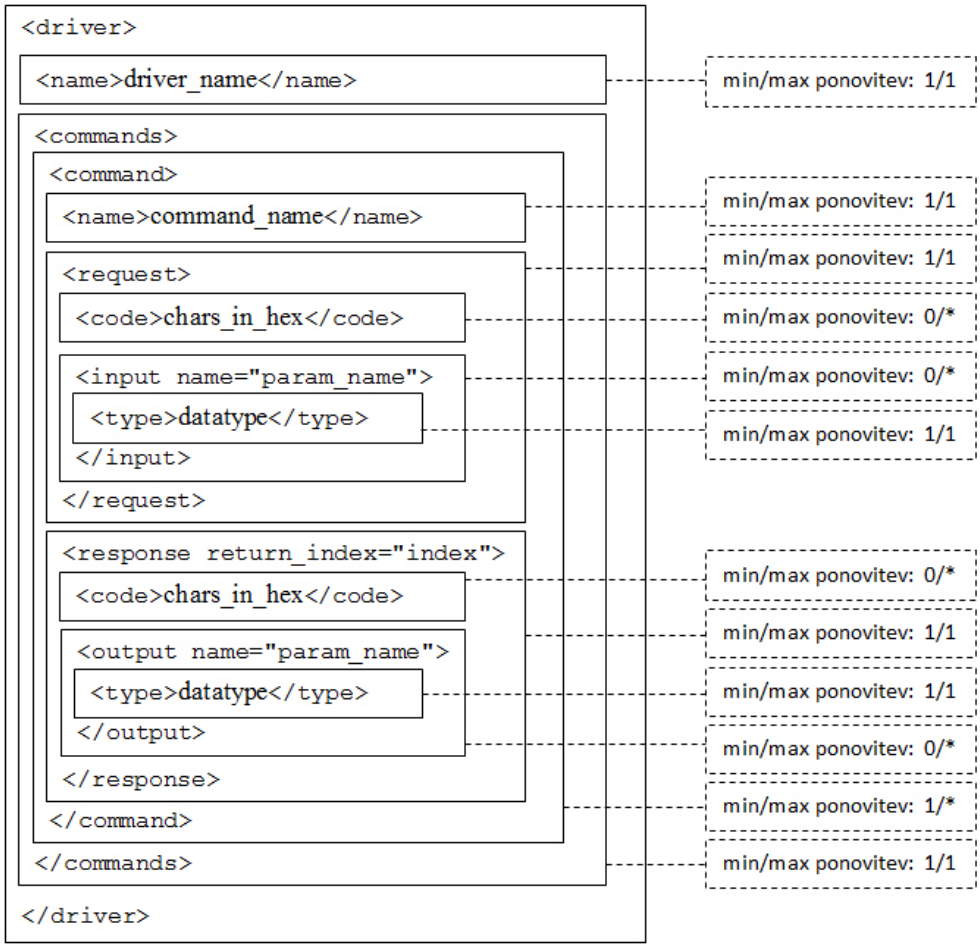
```
commandParser.parseResponse(commandID, &buffer[0], &values);
```

Končno funkcija element v namenskem vektorju vrednosti na mestu *responseNum* pretvori v podatkovni tip, specificiran pri klicu funkcije, pobriše vektor vrednosti in pretvorjeno vrednost vrne. V primeru napake pri pretvarjanju vrednosti, napake pri komunikaciji ali katerekoli druge napake, funkcija izstavi izjemo.

4.3.3 Konfiguracija

Vse potrebne informacije za programsko generiranje programske kode generičen modul pridobi iz konfiguracijske datoteke. Ta definira celotno podobo končnega gonilnika. Konfiguracijska datoteka mora biti značkovno strukturirana, tipa *xml*. Na sliki 4.1 je podana shema zahtevane strukture konfiguracijske datoteke.

Konfiguracijsko datoteko sestavlja ena glavna značka *driver*. Značka *driver* mora vsebovati natanko dve novi znački, *name* in *commands*. Značka *name* na tem mestu določa ime končnega gonilnika, značka *commands* pa s svojo strukturo določa vse funkcije in njihove operacije, ki jih bo končni



Slika 4.1: Struktura konfiguracijske datoteke

gonilnik vseboval. Značka *commands* je lahko sestavljena iz ene ali več novih značk *command*. Vsaka od teh definira eno operacijo in pripadajočo funkcijo končnega gonilnika. Vsaka značka *command* mora vsebovati značko *name*, ki na tem mestu definira ime funkcije, ki jo pripadajoča značka *command* opredeljuje. Nadalje mora vsaka značka *command* vsebovati natanko eno značko *request* in eno značko *response*. Prva definira format zahtevka za pripadajočo operacijo na napravi in vhodne parametre funkcije končnega gonilnika. Druga definira format odgovora, s katerim se naprava odzove na zahtevo, hkrati pa definira tudi vrednost, ki jo pripadajoča funkcija končnega go-

nilnika vrne. Značka *request* je lahko sestavljena iz poljubnega števila značk *code* in poljubnega števila značk *input*. Vrstni red značk znotraj značke *request* je pomemben, saj skupaj z vsebino značk definira format zahtevka. Vsaka značka *code* definira zaporedje znakov, ki so v formatu zahtevka, na istem mestu kot pozicija značke *code* znotraj *request*, statični. Značke *input* definirajo mesta v formatu zahtevka, kamor bodo vnešene vrednosti, ki jih pripadajoča funckija prejme kot vhodne parametre. S tem značke *input* definirajo tudi število in vrsto vhodnih parametrov dotične funckije. Vsaka značka *input* mora vsebovati atribut *name*, čigar vrednost definira ime pripadajočega vhodnega parametra funkcije. Poleg tega mora značka *input* vsebovati tudi značko *type*, ki opredeljuje podatkovni tip parametra. Trenutno podprti podatkovni tipi so *Float*, *Double*, *Short*, *Int*, *Long*, *String*, *Byte* in *Boolean*. Podobno kot *request*, lahko značka *response* vsebuje poljubno število značk *code* in *output*, katerih vrstni red je prav tako pomemben in sodoloča format odgovora. Značka *code* podobno definira zaporedje enega ali več znakov, ki so na pripadajočem mestu v odgovoru statični. Značke *output* so po strukturi enake značkam *input*, definirajo pa mesta v odgovoru, na katera naprava shrani vrednosti. Poleg tega definirajo tudi imena teh vrednosti ter njihove podatkovne vrste. Značka *response* lahko poleg vsega vsebuje tudi atribut *return_index*. Ta označuje index elementa *output*, ki naj ga pripadajoča funkcija končnega gonilnika vrne. V kolikor atribut *return_index* ni definiran ali pa je prazen, dotična funkcija ne bo vračala vrednosti.

4.3.4 Generiranje programske kode

Generiranje programske kode je osnovna funkcionalnost generičnega modula. Naloga programskega generiranja je, z uporabo predloge implementacije, opisane v prejšnjih poglavjih, in z uporabo priložene konfiguracijske datoteke, ustvariti implementacijo končnega gonilnika za neko določeno periferno napravo. Proces je sestavljen iz treh faz. V prvi fazi generičen modul prebere in razčleni konfiguracijsko datoteko. V drugi fazi, na podlagi pridobljenih informacij, generičen modul generira vso potrebno programsko kodo. V za-

dnji fazi modul predlogo implementacije gonilnika skopira, preimenuje, in vse nosilce v predlogi zamenja s kodo, generirano v drugi fazi. V nadaljevanju sledi natančnejši opis omenjenih faz.

Funkcionalnost generiranja smo implementirali v programskem jeziku *Python*. Zbrana je v sklopu skripte *generateDriver.py*, ki služi kot orodje uporabniku za ustvarjanje željenega gonilnika.

V implementaciji sktipte smo uporabili standardne *Python* knjižnice: *sys*, *os*, *xmle.etree*, *shutil* in *re*.

Skripta kot vhodni parameter pričakuje lokacijo konfiguracijske datoteke, na podlagi katere generira podatke. Končno izdelan gonilnik skripta namesti v podmapo z imenom, ki sovpada z imenom gonilnika, specificiranega v konfiguracijski datoteki. Bolj podrobno je postopek uporabe generičnega gonilnika opisan v naslednjem poglavju.

Faza1: Branje konfiguracije

V prvi fazi skripta preveri in prebere vhodni parameter, ki identificira lokacijo konfiguracijske datoteke. V drugem koraku podano konfiguracijsko datoteko prebere ter preveri strukturo, če je le ta veljavna. V sklopu prve faze skripta preveri tudi, če so prisotne vse datoteke predloge implementacije, ki se morajo nahajati v podmapi $./GenericModule/DefaultCode/$. Če v kateremkoli koraku pride do napake, skripta napako sporoči preko terminala in se ustavi.

Faza2: Generiranje kode

V drugi fazi skripta, po navodilih konfiguracijske datoteke, generira vso potrebno programsko kodo. Generira štiri sklope kode:

- NAME_GEN
- NAMEU_GEN
- FUNCTION_DECLARATIONS_GEN

- FUNCTION_DEFINITIONS_GEN
- COMMAND_ENUM_GEN
- COMMANDS_CREATE_GEN

V prvem koraku skripta iz prebrane vsebine konfiguracije izlušči ter loči vse kose informacij. Nekatere kose informacij sestavi iz ostalih. Nekateri kosi informacij so v tem koraku za namene lažjega sestavljanja končnih sklopov kode že generirani v obliko programske kode. Vsi različni kosi informacij so opisani v tabeli 4.5.

Tabela 4.5: Sestavni kosi informacij

Kosi informacij z opisom	
DRIVER_NAME	
	Ime gonilnika
COMMANDS	
	Skupek objektov tipa $COMMAND$, kjer vsaka nosi informacije o eni izmed funkcij gonilnika.
COMMAND	
	Objekt, ki zružuje vse informacije o eni izmed funcij gonilnika. Te so tipa $COMMAND\_NAME$, $COMMAND\_ID$, $COMMAND\_RETURN\_TYPE$, $COMMAND\_INPUTS$, $COMMAND\_INPUT\_FORMAT$, $COMMAND\_OUTPUT\_FORMAT$, $RESPOND\_CALL$, $LOAD\_CALLS$, $SEND\_CALL$ in $RESPONSE\_BUFFER\_LEN$.

Kosi informacij z opisom	
COMMAND_NAME	
	Ime dotične funckije.
COMMAND_ID	
	Identifikator dotične funkcije, generiran iz imena le te:
	$COMMAND\_ID = re.sub(r"(? <= \w)([A - Z])", r"_\1", COMMAND\_NAME).upper()$ Primer: ReadGain $->$ READ_GAIN
COMMAND_RETURN_TYPE	
	Tip vrednosti, ki jo dotična funkcija vrne.
COMMAND_RETURN_INDEX	
	Index elementa, ki naj ga dotična funkcija vrne.
COMMAND_INPUTS	
	String vrednost, ki vključuje tip in ime vsakega izmed vhodnih parametrov funkcije, ločenih z ",". String je generiran za dotično funkcijo gonilnika sledeče:
	$for\ input\ in\ inputs:$ $COMMAND\_INPUTS\ +=\ INPUT\_TYPE\ +\ "\ "\ +$ $INPUT\_NAME\ +","$
COMMAND_INPUT_FORMAT	
	Format ukaza operacije gonilnika, ki pripada dotični funkciji. Format se uporablja pri podajanju formatov za operacije v sestavljalnem modulu, in je temu priperno strukturiran:
	$for\ element\ in\ request:$ $if\ element.type == input:$ $COMMAND\_INPUT\_FORMAT+ = \%s$ $else: COMMAND\_INPUT\_FORMAT+ = element$
COMMAND_OUTPUT_FORMAT	
	Format odgovora operacije gonilnika, ki pripada dotični funkciji. Format se uporablja pri podajanju formatov za operacije v sestavljalnem modulu, in je temu priperno strukturiran:

Kosi informacij z opisom	
	$for\ element\ in\ response$: $if\ element.type == output$: $COMMAND\_OUTPUT\_FORMAT+ = \%s$ $else : COMMAND\_OUTPUT\_FORMAT+ = element$
RESPONDE_CALL	
	Klic generične funkcije *respond*, generiran za dotično funkcijo gonilnila:
	$RESPONDE\_CALL = "\backslash treturn\ Respond < " +$ $COMMAND\_RETURN\_TYPE+»("+COMMAND\_ID+","+$ $RESPONSE\_BUFFER\_LEN+","+str(RETURN\_INDEX)+$ $");\backslash n"$
LOAD_CALLS	
	Serija klicev generične funkcije *load*, generiranih za dotično funkcijo gonilnika:
	$for\ input\ in\ inputs$: $LOAD\_CALLS+ = "\backslash tLoad < " + INPUT\_TYPE + »(" +$ $INPUT\_NAME + ");\backslash n"$
SEND_CALL	
	Klic generične funkcije *send*, generiran za dotično funkcijo gonilnila:
	$SEND\_CALL = "\backslash tSend(" + COMMAND\_ID + ");\backslash n"$
RESPONSE_BUFFER_LEN	
	String, ki v obliki $C++$ progrmaske kode drži enačbo za izračun potrebne dolžine bufferja pri prevzemu odgovora naprave, generiran namensko za dotično funkcijo:
	$RESPONSE\_BUFFER\_LEN = "(0"$ $foroutput\ in\ outputs$: $RESPONSE\_BUFFER\_LEN\ += "\ +\ sizeof(" +$ $OUTPUT\_TYPE + ")"$

Kosi informacij z opisom

V naslednjem koraku skripta uporabi zgrajene kose za sestavljanje navedenih celotnih sklopov programske kode. Proces sestavljanja končnih sklopov kode:

- Sklop kode: NAME_GEN

```
NAME_GEN = DRIVER_NAME
```

- Sklop kode: NAMEU_GEN

```
NAMEU_GEN = NAME_GEN.upper()
```

- Sklop kode: FUNCTION_DECLARATIONS_GEN

```
for COMMAND in COMMANDS:
    FUNCTION_DECLARATIONS_GEN +=
        "\t" + COMMAND.COMMAND_RETURN_TYPE + " " +
        COMMAND.COMMAND_NAME + "(" + COMMAND.COMMAND_INPUTS +
        ") throw(" + DRIVER_NAME + "Exception);\n"
```

- Sklop kode: COMMAND_ENUM_GEN

```
COMMAND_ENUM_GEN = "enum command_name_t{\n"
for COMMAND in COMMANDS:
    COMMAND_ENUM_GEN += "\t" + COMMAND.COMMAND_ID + ",\n"
COMMAND_ENUM_GEN += "\t\tCARRIAGE_RETURN\n};"
```

- Sklop kode: COMMANDS_CREATE_GEN

```
for COMMAND in COMMANDS:
    COMMANDS_CREATE_GEN +=
        "\t\tcommand_formats.push_back((command_format_t){" +
        COMMAND.COMMAND_ID + ",\"" +
        COMMAND.COMMAND_INPUT_FORMAT + "\"" + ",\"" +
        COMMAND.COMMAND_OUTPUT_FORMAT + "\"" + "});\n"
```

- Sklop kode: FUNCTION_DEFINITIONS_GEN

```
for COMMAND in COMMANDS:
    FUNCTION_DEFINITIONS_GEN += COMMAND.COMMAND_RETURN_TYPE +
        " " + NAME_GEN + "Driver::" + COMMAND.COMMAND_NAME +
        "(" + COMMAND.COMMAND_INPUTS + ") throw(" + NAME_GEN +
        "Exception)\n{\n \tMassFlowGuard guard = " +
        "MassFlowGuard(&read_mutex);\n" + COMMAND.LOAD_CALLS +
        COMMAND.SEND_CALL + COMMAND.RESPOND_CALL + "}\n\n"
```

Faza3: Sestavljanje končnega gonilnika

V tretji fazi programskega generiranja kode, generični modul sestavi končni gonilnik. V prvem koraku skripta preveri predlogo implementacije gonilnika. Ta se mora nahajati v podmapi *./GenericModule/DefaultCode/*. V naslednjem koraku ustvari novo podmapo z imenom, ki je enak imenu gonilnika, specificiranega v konfiguracijski datoteki, kjer imenu na koncu doda "*Driver*". V kolikor podmapa z istim imenom že obstaja, skripta to sporoči preko ukazne vrstice in se ustavi. Po uspešno ustvarjeni podmapi, skripta vso predlogo implementacije gonilnika skopira vanjo, pri čemer imena datotek preimenuje tako, da znak "_"v imenu datotek zamenja z imenom gonilnika, podanega v konfiguraciji.

V zadnjem koraku skripta odpre vse predhodno ustvarjene datoteke končnega gonilnika, zapovrstjo. V vsakem od teh poišče nosilce vrednosti, podane v tabeli 4.4. Nosilce vrednosti v vsaki datoteki zamenja s pripadajočimi programsko generiranimi sklopi kode, generiranimi v drugi fazi, in jih shrani. Preslikava nosilcev vrednosti v sklope programske kode je opisana v tabeli 4.6.

Sklop ustvarjenih datotek predstavlja končni, specifični gonilnik, ustvarjen programsko, izključno po navodilih podane konfiguracije.

Tabela 4.6: Preslikava nosilcev vrednosti v programsko kodo

Nosilci vrednosti	Sklopi programske kode
$NAME$	NAME_GEN
$NAMEU$	NAMEU_GEN
$FUNCTION_DECLARATIONS$	FUNCTION_DECLARATIONS_GEN
$FUNCTION_DEFINITIONS$	FUNCTION_DEFINITIONS_GEN
$COMMAND_ENUM$	COMMAND_ENUM_GEN
$COMMANDS_CREATE$	COMMANDS_CREATE_GEN

4.4 Uporaba gonilnika

Uporaba generičnega gonilnika je preprosta. Izvaja se v dveh korakih:

- 1. Pisanje konfiguracijske datoteke

  Uporabnik mora, po navodilih, opisanih v prejšnjih poglavjih, kreirati konfiguracijsko datoteko, da ta odraža željeno funkcionalnost gonilnika.

- 2. Izvršitev skripte za generiranje gonilnika:

  ```
  # python generateDriver.py PATH_TO_CONFIGURATION
  ```

  Vhodni parameter $PATH\_TO\_CONFIGURATION$ mora kazati na konfiguracijsko datoteko, ustvarjeno v prvem koraku.

Pri pravilni strukturi konfiguracije bo po izvršenem drugem koraku specifični gonilnik ustvarjen v podmapi z imenom, ki sovpada z njegovim imenom. Končnemu gonilniku pripadata tudi komunikacijski in sestavljalni modul.

Poglavje 5

Zaključek

Cilj diplomske naloge, to je poenostavitev implementacije gonilnikov, je z predstavljenimi rešitvami mogoč. Kot rezultat naloge je predstavljen generičen gonilnik. Ta omogoča hitro razširitev na želen specifičen gonilnik, ki je takoj pripravljen za vse potrebe testiranja. Poleg tega razvit generičen gonilnik podaja osnovo nadaljnim študijam za še večjo avtomatizacijo pri implementaciji in širšo podporo pri zahtevah delovanja. V okviru naloge smo predpostavili omejitev pri asinhroni komunikaciji, ki bi jo v prihodnje lahko razvili.

Literatura

[1] R.K. Agrawal and V.R. Mishra. The design of high speed uart, 2013.

[2] Hiroki Asai and Takeshi Hiraoka. Information processing apparatus, serial communication system, method of initialization of communication therefor, and serial communication apparatus, 2013. US Patent App. 13/646,262.

[3] Klaus Brandstätter, Stefan Heinrich, and Simon Vella. System and method for controlling multiple computer peripheral devices using a generic driver, 2013.

[4] Shuangye Chen, Dongyue Zheng, and Rujun Yang. Design of preverifying serial-ethernet communication gateway based on lm3s8962, 2013.

[5] R. Dobkin, M. Moyal, A. Kolodny, and R. Ginosar. Asynchronous current mode serial communication, 2010.

[6] Shu Leng Dong. Study of serial-communication in home environment control system, 2013.

[7] Robert Horning, Renffeh, DavidCary, Jhdiii, HumbertoDiogenes, Netch, Breakpoint, Insaneinside, Ashfaq, and Gareth Richardson. *Serial Programming*. Wikibooks, 2013.

[8] Xiao Hua Sun, Li Yan, Zhi Yong Zhang, and Fu Shun Wang. Serial communication design of industrial control system based on visual basic6.0 and mx software package, 2013.

[9] Nai Ning Wen, Shang Fu Gong, and Yan Jun Wang. The implementation of serial communication cserial the port class, 2013.

[10] Zhenghua Xin, Hongmei Lu, Liangyi Hu, and Jianxin Li. Implementation of spi and driver for cc2430 and c8051f120, 2012.

[11] Tong Zhang, Jun hai Jiang, and Nian feng Li. Study on the application of serial communication programming technology in embedded system, 2012.